Extending R packages to support 64-bit compiled code: an illustration with spam64 and GIMMS NDVI$_{3g}$ Data

Furrer, Reinhard ; Gerber, Florian ; Mösinger, Kaspar

Abstract: Software packages for spatial data often implement a hybrid approach of interpreted and compiled programming languages. The compiled parts are usually written in C, C++, or Fortran, and are efficient in terms of computational speed and memory usage. Conversely, the interpreted part serves as a convenient user-interface and calls the compiled code for computationally demanding operations. The price paid for the user friendliness of the interpreted component is—besides performance—the limited access to low level and optimized code. An example of such a restriction is the 64-bit vector support of the widely used statistical language R. On the R side, users do not need to change existing code and may not even notice the extension. On the other hand, interfacing 64-bit compiled code efficiently is challenging. Since many R packages for spatial data could benefit from 64-bit vectors, we investigate strategies to efficiently pass 64-bit vectors to compiled languages. More precisely, we show how to simply extend existing R packages using the foreign function interface to seamlessly support 64-bit vectors. This extension is shown with the sparse matrix algebra R package $_{spam}$. The new capabilities are illustrated with an example of GIMMS NDVI$_{3g}$ data featuring a parametric modeling approach for a non-stationary covariance matrix.

DOI: https://doi.org/10.1016/j.cageo.2016.11.015

Computers & Geosciences 104 (2017) 109–119

Contents lists available at ScienceDirect

Computers & Geosciences

journal homepage: www.elsevier.com/locate/cageo

Extending R packages to support 64-bit compiled code: An illustration with spam64 and GIMMS $NDVI_{3g}$ data

Florian Gerber[a], Kaspar Mösinger[a], Reinhard Furrer[b,*]

[a] Department of Mathematics, University of Zurich, Switzerland
[b] Departments of Mathematics and Computational Science, University of Zurich, Switzerland

ARTICLE INFO

Keywords:
Sparse matrix
Non-stationarity
Compactly supported covariance function
Huge dataset
dotCall64
Foreign function interface

ABSTRACT

Software packages for spatial data often implement a hybrid approach of interpreted and compiled programming languages. The compiled parts are usually written in C, C++, or Fortran, and are efficient in terms of computational speed and memory usage. Conversely, the interpreted part serves as a convenient user-interface and calls the compiled code for computationally demanding operations. The price paid for the user friendliness of the interpreted component is—besides performance—the limited access to low level and optimized code. An example of such a restriction is the 64-bit vector support of the widely used statistical language R. On the R side, users do not need to change existing code and may not even notice the extension. On the other hand, interfacing 64-bit compiled code efficiently is challenging. Since many R packages for spatial data could benefit from 64-bit vectors, we investigate strategies to efficiently pass 64-bit vectors to compiled languages. More precisely, we show how to simply extend existing R packages using the foreign function interface to seamlessly support 64-bit vectors. This extension is shown with the sparse matrix algebra R package `spam`. The new capabilities are illustrated with an example of GIMMS $NDVI_{3g}$ data featuring a parametric modeling approach for a non-stationary covariance matrix.

1. Introduction

This research addresses the handling of very large vectors in R, and in our case was motivated through huge covariance matrices resulting from dependencies of georeferenced data, but any other scientific domain handling very large datasets could have served as motivation.

Spatial statistics relies on modeling the first *and* second order structures of directly observed or latent spatial fields. Typically, only one realization of such a spatial field is observed, and therefore parametric models for the second order structure is the prime choice. For maximum likelihood estimation or prediction (through classical kriging or other means) the covariance matrix of the spatial field has to be explicated. While the construction thereof is typically feasible, the operations based on these matrices (solving linear systems and calculating determinants) are the computational bottlenecks. Dataset sizes that can be dealt with a brute force implementation are on the order of several thousands—essentially the same size as a decade ago. However, with a careful model design, it is now possible to handle spatial datasets on the order of 10^5 to 10^6 on typical computing machines. These approaches can be classified into roughly two different categories. A model for which an efficient implementation exists (Kronecker formulation, separable models, e.g., Genton, 2007; Furrer and Genton, 2011) or for which an approximation is available (tapering, e.g., Furrer et al., 2006, Kaufman et al., 2008; Furrer et al., 2016; low-rank models, e.g., Cressie and Johannesson, 2008; Banerjee et al., 2008; Stein, 2008, composite likelihood approaches, e.g., Stein et al., 2004; Bevilacqua et al., 2012; Eidsvik et al., 2014, Gaussian Markov random fields type approximations, e.g., Hartman and Hössjer, 2008; Lindgren et al., 2011, etc.). For a review of statistical approaches for large datasets, see Sun et al. (2012).

While datasets on the order of 10^5 to 10^6 seem large, they are still much smaller than a typical Landsat 7 satellite image, which consists of more than 34 million pixels (30 m resolution for an approximate scene size of 170 km×183 km; source landsat.usgs.gov). Fitting spatial models on this data is challenging and limited by the available computing resources. Surprisingly the limiting factors are in some cases RAM and not the performance of the CPU(s). This is even more an issue when the computing software does not exploit the entire available memory. Until recently, the successful open source software R was bound to 32-bit addressing and thus limited the size of matrices independent of the available RAM. This limit implies that all (atomic) vectors have to have less than $2^{31} \approx 2.147 \cdot 10^9$ elements. At first sight

* Corresponding author.
E-mail address: reinhard.furrer@math.uzh.ch (R. Furrer).

http://dx.doi.org/10.1016/j.cageo.2016.11.015
Received 4 October 2015; Received in revised form 21 November 2016; Accepted 30 November 2016
Available online 07 December 2016

this seems huge, however a covariance matrix of a 160×320 lat/lon grid cannot be managed (corresponding to T106 spectral grid resolution climatedataguide.ucar.edu/climate-model-evaluation/common-spectral-model-grid-resolutions). Naturally, using sparse matrices (through, e.g., tapering) larger datasets are possible. However this limit is easily overdrawn with Landsat 7 satellite images and recommended taper ranges.

This article focuses on georeferenced data, but also the analysis of other data types is limited by the 32-bit constraint of R. One example from the authors recent research is the modeling of the covariance structure of microarray data with roughly 1.4 million probe sets on nowadays arrays https://www.affymetrix.com/catalog/131452/AFFY/Human+Exon+ST+Array, Furrer and Sain, 2009).

With the recent release of the R version 3.0.0, basic operations have been extended to be able to handle 64-bit vectors. However, it is not possible to directly pass long vectors from R to compiled code containing 64-bit integers though the foreign function interface. Although efforts exist to simplify the integration of compiled code in R (e.g., Eddelbuettel et al., 2016; Eddelbuettel, 2013), we are not aware of any interface that simplifies the interaction with 64-bit compiled code. This is unfortunate because there are many packages available for spatial data that relay on compiled code and could benefit from an extension to long vectors; for example, see the CRAN task views "Analysis of Spatial Data" (Bivand, 2016) and "Handling and Analyzing Spatio-Temporal Data" (Pebesma, 2016). This article sheds light on how to extend existing R packages with 64-bit compiled code and we will refer to such R packages as "64-bit packages." Since one can think of many possible approaches to cope with the 64-bit issue, we tried to find a strategy that has the following features: (i) From the end user perspective the enhanced 64-bit package should first of all cover all the functionality that was available before the extension without any performance losses in terms of memory usage and speed. Existing R code should be portable to the 64-bit package without any changes. Furthermore, the user should not be force to think about storage modes of vectors. (ii) From the developer perspective the work to migrate a package to 64-bit as well as the maintenance time should be kept at a minimum.

This article is structured as follows. Section 2 gives some background on how to call 64-bit compiled code from R. After covering general ideas and concepts, some technical details are given (a section that can be skipped). Finally, we introduce the R package `dotCall64` which simplifies the call to compiled code with 64-bit integers. Section 3 shows how to use both a 32-bit and a 64-bit version of the compiled code such that for small problems no computational and storage losses occur. Readers that are mainly interested in analyzing spatial data using huge sparse matrices are referred to Sections 4 and 5. In Section 4 we illustrate the porting of `spam`, an existing package to manage sparse matrices, to 64-bit capability and show the user relevant aspects with examples and performance measurements. In Section 5 we model the covariance of a GIMMS $NDVI_{3g}$ residual field involving "64-bit" Cholesky factors as a proof of concept. In Section 6 we conclude with a short discussion and outlook. The package `spam` and the R scripts that were used to create the figures and tables of this article are available at https://github.com/florafauna/CAGEO-spam64-supplement. A current development version of `spam` is available in the git repository https://git.math.uzh.ch/reinhard.furrer/spam.

2. Calling compiled code with 64-bit integers from R

2.1. General ideas and concepts

We now shed some light on the 64-bit implementation of R. While the focus of this section is the general concept, more technical insights are given in the next section. In R, vectors are one of the most basic object types. They can be thought of as a string of many elements that can be indexed according to their (relative) position. The indexing is based on (signed) 32-bit integers and thus the length of vectors is limited to $2^{31} - 1$ elements. Starting from release 3.0.0 in early 2013, basic support for vectors up to size 2^{52} is supplied; see also Section 12 on https://cran.r-project.org/doc/manuals/r-devel/R-ints.html. These vectors (including raw, logical, integer, numeric and character vectors, and lists and expression types) are called *long vectors*. The R implementation is such that for long vectors, doubles are used for addressing and minor modifications are required for the function `length()`, which returns a double in the case of long vectors. The extension has been done without breaking existing code and thus some of the implementation seems at first sight suboptimal. Notice that in R, matrices or general type arrays are objects where the data are stored in a vector and which possess a dimension attribute. Hence, the above-mentioned construction of long vectors still applies.

For efficient use, packages now have to supply the possibility of handling long vectors as well, and thus the underlying C/C++ or Fortran code has to be compiled in 64-bit mode. While the addressing in the compiled code is typically done with (signed) 64-bit integers, the discrepancy between the compiled and interpreted component are apparent.

There are several approaches to cope with this discrepancy in the storage mode and the two main ones are: (1) rewrite compiled code and use doubles instead of integers. (2) use doubles on the R side and cast them to 64-bit integers before calling the compiled code. The former requires a big effort on the package maintainers to rewrite existing code. Additionally, in the case R changes implementation to long integer addressing, many changes in the source code of the packages are required. The latter can be handled though an additional function that handles the type conversions (also denoted with casting) from double to 64-bit integers and back again. While this approach does not require any changes in existing compiled code, it implies a slight performance loss as casting between the storage modes takes time. We have evaluated the two approaches as well as additional flavors thereof (Mösinger, 2015) and chose the second approach.

Throughout the paper we will use the term "32-bit integers" to refer to the integer type in R and the 32-bit integers in the compiled code. On the other hand "64-bit integers" refer to doubles in R and 64-bit (long) integers in the compiled code.

2.2. Technical implementation

This section gives some technical insights into the underlying C implementation of the long vector support of R and can be skipped without loss of the general idea. We will refer to the R source code of version 3.1.0 (Core Team, 2016a) in several places by indicating the path to the source file and the line number. The mentioned files are available in the supplementary material of this paper. In addition, we highlight changes in the source files `src/include/Rinternals.h`, `src/include/Rinlinedfuns.h`, and `src/main/memory.c`, which were made to support long vectors. They are given in the files `diff_Rinternals.h`, `diff_Rinlinedfuns.h`, and `diff_memory.c` of the supplementary material and are the result of `svn diff -r 59004:59009` of the corresponding files in the R svn repository (http://svn.r-project.org/R/trunk/).

Long vectors have been introduced without breaking existing code. For example, the widely used C function `R_len_t length(SEXP s)` (defined in `src/include/Rinlinedfuns.h:122`) returns the length of a SEXP (S expression) as a `R_len_t`, which typedef'ed as `int32_t` (defined in `src/include/Rinternals.h:49`). Any code that assumes that `length(SEXP s)` returns an `int32_t` is compatible with this declaration. However, if `SEXP s` is a *long vector* and therefore the length cannot be stored inside an `int32_t`, the operation returns an error (See `src/main/memory.c:3828` which is called at `src/include/Rinternals.h:325`). Therefore, any legacy code that calls `R_len_t length(SEXP s)` still works on *short vectors* and does not need to be changed.

To get the length of *long vectors*, one has to call the newly defined function `R_xlen_t xlength(SEXP s)` (defined in `src/include/Rinlinedfuns.h:154`) instead. If R is compiled with *long vector* support, `R_xlen_t` is typedef'ed as `ptrdiff_t` (defined in `src/include/Rinternals.h:62`), where "`ptrdiff_t` is the signed integer type of the result of subtracting two pointers. This will probably

be one of the standard signed integer types (short int, int or long int), but might be a nonstandard type that exists only for this purpose." (The GNU C Library GNU, 2014, A.4 Important Data Types.)

We now sketch how *long vectors* are actually implemented. In R, vectors are made out of a header of type `VECSEXP` (defined in `src/include/Rinternals.h:273`) that is followed by the actual data. The header contains a field `length` of type `R_len_t` and hence cannot capture the length of a *long vector* as we have seen previously. Whenever the actual length is larger than $2^{31} - 1$, the length inside `VECSEXP` is set to -1 and an additional header of type `R_long_vec_hdr_t` is prefixed, which contains a field called `length` of type `R_xlen_t`.

The current implementation of R defines the R-type integer as signed 32-bit `int` (known as `int32_t`). There is no other integer type; there is no `R_xlen_t` equivalent in R. Instead, any integer number greater than $2^{31} - 1$ is stored as a double, which is integer-precise up to about 2^{52} (see `help("long vectors")`). As a consequence (1) *long vectors* are indexed by doubles and (2) if the length of a vector is larger than 2^{31} - 1 the R function `length(x)` returns a number of type double (see `help(length)`).

Note that the R package `bit64` (Oehlschlägel, 2015) provides a more efficient data type for 64-bit integers compared to the 64-bit integer provided by R. However, the package does not support *long vectors* and thus cannot be used in our context.

2.3. Alternative interface provided by the package `dotCall64`

Mösinger (2015) implemented an extension of R's foreign function interface, which is available in the R package `dotCall64` (Mösinger et al., 2016). The package provides an interface (written in C, exposed as an R function) that can be used to call compiled code in a way that (i) the arguments of the function are copied if and only if necessary (ii) 64-bit integers are cast from double (the R storage mode of 64-bit integers) to 64-bit integers before calling the compiled code and cast from 64-bit integers to doubles afterwards again (iii) supports long vectors.

Next, we illustrate how the `dotCall64` package can be used to call compiled code. Assume a hypothetical Fortran function `fun` that takes one integer argument `arg`. (The example would be similar for a C/C++ function.) Given the function is properly compiled and loaded in R, it can be called with the integer argument `arg=1L` via

```
R> out <- .Fortran("fun", arg)
```

The same call can be made via the R function `.C64()` from the R package `dotCall64`, which is available on CRAN.

```
R> install.packages("dotCall64")
R> library("dotCall64")
R> out <- .C64("fun", SIGNATURE = "int", arg)
```

Here, we additionally have to specify the argument `SIGNATURE`, which is set to integer in this case. In this situation, the result is the same as with the `.Fortran()` call. The main advantage of using `dotCall64` becomes obvious when `fun` is changed such that it takes a 64-bit integer as argument. If we now set `arg=2^32` and call the function via `.Fortran("fun", arg)` the Fortran code will interpret the 8 bytes of the double as a 64-bit integer, likely resulting in a crash. On the other hand, the call via `.C64()` can be adapted to expect a 64-bit integer by setting the argument `SIGNATURE` to `"int64"`. With that the call returns the desired result.

```
R> out <- .C64("fun", SIGNATURE = "int64", arg)
```

Instead of using the R function `.C64()`, the C function `dotCall64()` of the R package `dotCall64` can be called directly. This is especially useful when the compiled code relies on the C API of R or extensions thereof like, e.g., the R package `Rcpp` (Eddelbuettel et al., 2016; Eddelbuettel and François, 2011; Eddelbuettel, 2013). More detailed information on `dotCall64` including a description of the implementation and more extensive examples is given in Gerber et al. (2016).

3. Managing 32-bit and 64-bit compiled code

3.1. Motivation and general framework

As described in the previous section, the `dotCall64` interface takes care of the necessary casting when calling compiled code with 64-bit integer vectors from R. Hence, one could create an R package using doubles (instead of integers) on the R site and 64-bit integers in the compiled code. The drawback of this approach is that type conversion takes time and using 64-bit integers instead of integers is a waste of memory when the same could be done with integers. To illustrate this consider the extraction of one element out of a vector of length 2^{30} through compiled code. In the case where the vector is of type integer this operation is virtually instantaneous (order of milliseconds). On the other hand, if the same vector is stored as doubles (the format of a 64-bit integer in R) the same operation requires 4 Gb of additional physical RAM and takes about two seconds because of the necessity of casting from double to 64-bit integers. This motivates the uses of integers in R and calls compiled code with 32-bit integers whenever possible, or equivalently, uses the 64-bit variants only if necessary. Hence, the compiled code needs to be provided with both 32-bit integers and 64-bit integers.

We implemented the work flow of an R function calling potentially 64 bit compiled code as schematically illustrated in Fig. 1. The input of the R function may contain integers or 64-bit integer vectors, besides vectors of other types. After preprocessing the arguments in R, it is decided whether to use the compiled code with 32-bit integers or 64-bit integers. The compiled code is then called though a function from `dotCall64` that does the copying and casting of the arguments if necessary. Back in R, the results are post-processed and it is decided whether to returned the integer vectors as integers or doubles to R.

3.2. An S4 class to handle 32-bit and 64-bit integer vectors

As mentioned above it is beneficial in terms of performance to use the (32-bit) integer R type whenever possible to store integers, and doubles otherwise. This gets more involved when using S4 classes to

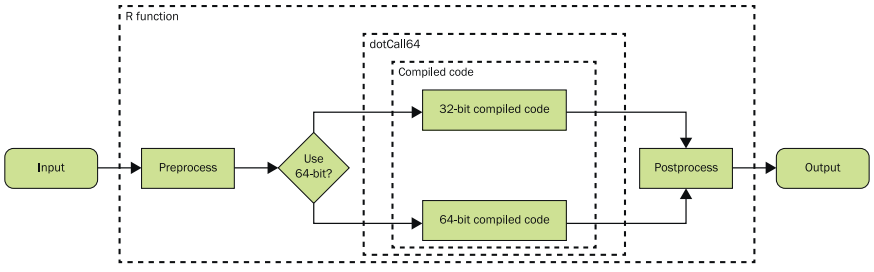

Fig. 1. Flow diagram of an R function calling compiled code as proposed in Section 3.1: First the arguments are preprocessed. Then it is decided whether to use the compiled code with 32-bit or 64-bit integers. Then the compiled code is called through functions provided by `dotCall64`, which takes care of copying and casting the arguments if required. In a post processing the step the results are collected and returned to R, having integers stored as integers or as doubles (64-bit integers).

store the data. To illustrate this we consider a class `simple` having one slot `entry` of type integer.

```
R> simple <- setClass(Class = "simple",
+    slots = c(entry = "integer"))
```

Since R stores 64-bit integer as doubles, this class cannot be used to store long vectors. Extending the class to accept long vectors efficiently while reassuring backward compatibility and maintainer friendliness is not trivial. Mösinger (2015) experimented with various designs. One idea is to let the class `simple` remain unchanged and define a new class `simple64` that is used to store 64-bit integers. In this situation one could, e.g., add an additional constructor function that decides whether to create an object of class `simple` or `simple64` or even overwrite the constructor of `simple` with that constructor. The disadvantage of this design is that we have to manage two S4 classes.

Another design that uses only one S4 class is illustrated with more details next. First, we modify the definition of the class `simple` to accept vectors of the class numeric.

```
R> simple <- setClass(Class = "simple",
+    slots = c(entry = "numeric"))
```

Note that the class numeric extends the class integer, and as a consequence, the slot `entry` accepts vectors of type double or integer. The decision to use one or the other type can be made in the initialize method of the class that we define as follows:

```
R> simple.init <- function(.Object, entry) {
+     if(max(abs(entry)) < .Machine$integer.max)
+          .Object@entry <- as.integer(entry)
+     else .Object@entry <- entry
+     return(.Object)
+ }
R> setMethod("initialize",
+    signature(.Object = "simple"),
+    simple.init)
[1] "initialize"
```

To illustrate the functionality of this class, we define the function `mult()`, which corresponds to a scalar multiplication of the class simple.

```
R> mult <- function(obj, factor)
+     return(simple(
+     entry = obj@entry * factor[1]))
```

Next, we create an instance of class simple:

```
R> print(s1 <- simple(entry = 2^29))
An object of class "simple"
Slot "entry":
[1] 536870912
```

Since 2^{29} is smaller than `.Machine$integer.max` (2^{31} in our environment) the slot `entry` is of type integer.

```
R> typeof(s1@entry)
[1] "integer"
```

When applying the function `mult()` to `s1` the appropriate storage format of the slot `entry` is chosen automatically.

```
R> s2 <- mult(s1, 8)
R> typeof(s2@entry)
[1] "double"
R> s3 <- mult(s2, 1/4)
R> typeof(s3@entry)
[1] "integer"
```

Note that the `initialize()` function is only called if the class constructor is called. Hence, the slot `entry` of the class can be overwritten without checking of the format via

```
R> s1@entry <- 1L
```

This may beneficial in terms of performance in some cases.

3.3. Code organization in two R packages

Now we have all necessary pieces together to extend an R package with 64-bit compiled code. At first sight, the organization of the code of such a package seems challenging from an R package developer point of view. Therefore, we give some insights in a code organization and development framework that reduces the additional work to a minimum.

Suppose we have a hypothetical package called `simplePkg` with 32-bit integers C/C++ or Fortran code called through the foreign function interface. To extend this package such that the functions can also be called with 64-bit integers we make use of the

Table 1
Distribution of code in two hypothetical R packages `simplePkg` and `simplePkg64`. The package `simplePkg` works with 32-bit compiled code and can be used independently of the other package. `simplePkg64` can be loaded as an add-on and enables the support for 64-bit vectors.

Package name	R code	Compiled code	manual
`simplePkg`	✓	✓(32-bit)	✓
`simplePkg64`	–	✓(64-bit)	–

Table 2
Code management in two hypothetical R packages `simplePkg` and `simplePkg64`. The first and third column summarize the basic file structure of the package. The middle column highlights the essential, minimal differences between the files, where <… indicates many more lines or files.

`simplePkg`		`simplePkg64`
`DESCRIPTION`	> `Package: simplePkg64` > `Depends: simplePkg` < `Package: simplePkg`	`DESCRIPTION`
`NAMESPACE`	> `useDynLib(simplePkg64)` < `useDynLib(simplePkg)` <…	`NAMESPACE`
`src`	Identical content for the source files. > The file `Makevars` containing (additional) compiler flags such as `PKG_FCFLAGS=-fdefault-integer-8`.	`src`
`man`	> Single file giving a short package overview <…	`man`
...	< More directories like `R`, `data`, `demo`, ...	

NAMESPACE feature of R. In R each package has a NAMESPACE, which allows the user to load different packages providing differently compiled functions with the same name. An argument in the foreign function interface (or in the dotCall64 functions) is then used to specify the package name and hence the compiled function is uniquely specified. This motivates the creation of an additional package simplePkg64 that contains the same source code for compiled code as in the package directory simplePkg/src/ but with 64-bit integers. This can be achieved with a simple call to GNU sed (2010) replacing all integer type declarations with "integer(8)". Alternatively, the GNU Fortran (2014) supports the flag -fdefault-integer-8, which can be set in simplePkg64/src/Makevars to declare integers as 64-bit integers. The remaining files and directories of the simplePkg64 are basically empty or reduced to a minimum. See Table 1 for a rough overview and Table 2 for a more detailed description of the (dis)similarities between the two packages. With this design, the package simplePkg works with 32-bit compiled code and can be used independently. Loading the simplePkg64 as an add-on enables the support of 64-bit integer vectors. We successfully tested the proposed strategy on Linux and Windows platforms.

Since the source code of the compiled code is the same in both packages, the additional time to maintain two packages instead of one is small. In fact, it is straightforward to design a Makefile that builds simplePkg and simplePkg64 out of one single package simplePkg.

4. Extending spam with 64-bit integer pointers

4.1. spam in a nutshell

We will now apply the ideas to the R package spam, which is an R package for sparse matrix algebra with emphasis on a Cholesky factorization of sparse positive definite matrices (Furrer and Sain, 2010). The implementation of spam is based on the competing philosophical maxims to be competitively fast compared to existing tools and to be easy to use, modify and extend. The first is addressed by using fast Fortran routines and the second by assuring S3 and S4 compatibility. One of the features of spam is to exploit the algorithmic steps of the Cholesky factorization and hence to perform only a fraction of the workload when factorizing matrices with the same sparsity structure. Simulations show that exploiting this break-down of the factorization results in a significant speed-up (Furrer and Sain, 2010).

To store the non-zero elements, spam essentially uses the "old Yale sparse format" (Eisenstat et al., 1977). In spam, a (sparse) matrix is stored as a S4 object with four slots (vectors), which are (1) the nonzero values row by row, (2) the ordered column indices of nonzero values, (3) the position in the previous two vectors corresponding to new rows, given as pointers, and (4) the column dimension of the matrix. Hence, to store a matrix with z nonzero elements spam requires z doubles and $z + n + 2$ integers compared to $n \times n$ doubles. Given the 32-bit limitation, we have the limit of (1) at most $2^{31} - 2$ rows, (2) at most $2^{31} - 1$ columns and (2) at most 2^{31} - 1 non-zero entries. More details about spam can be found in (Furrer and Sain, 2010; Gerber and Furrer, 2015).

4.2. Illustration of the functionality using spam64

The package spam64 is an implementation of the concept outlined in Section 3. It is based on spam version 2.x, which extends lower versions by modified Fortran calls and appropriate initializer methods. To enable 64-bit capability both the spam and the spam64 R packages need to be loaded (the latter depending on the former; see Table 2).

```
R> library("spam64")
R> grep(search(), pattern = "spam", value = TRUE)
[1] "package:spam"      "package:spam64"
```

As discussed above, the same top level R code for 32-bit and 64-bit matrices is used. Moreover, the user will notice the actual format only when looking explicitly at the storage format of the integer slots or by calling the print method of the spam object. As illustrated below, the functions return a spam object with 32-bit integers if possible and an object with 64-bit integers otherwise.

```
R> options(max.print = 14)
R> print(s1 <- spam(1:2^30))
Matrix of dimension 1073741824x1 with (row-wise)
nonzero elements:

[1]  1 2 3 4 5 6 7 8 9 10 11 12 13 14
[ reached getOption("max.print") -- omitted 1073741810
entries ]
Class 'spam' (32-bit)
R> print(s2 <- cbind(s1, s1))
Matrix of dimension 1073741824x2 with (row-wise)
nonzero elements:

[1] 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8
[ reached getOption("max.print") -- omitted 2147483633
entries ]
Class 'spam' (64-bit)
R> s2[1:2,]
     [,1] [,2]
[1,]    1    1
[2,]    2    2
Class 'spam' (32-bit)
```

In this R output, "(32-bit)" means that the slots colindices, rowpointers and dimension of the spam64 object are of type integer, as opposed to "(64-bit)" where these slots are of type double. The user may also force spam64 to return an object with slots of type double by setting the global option options(spam.force64=TRUE) or by setting the argument force64 of a specific function call to TRUE, e.g.,

```
R> spam(1, force64 = TRUE)
     [,1]
[1,]    1
Class 'spam' (64-bit)
```

4.3. Performance measurements

To get an impression of the performance in terms of speed and memory usage of spam64, we compared the implementation with spam and the matrix class from the base package using the following test setup: Matrices of dimension 2000×2000 with different percentages of randomly placed and randomly generated non-zero entries were generated. If the function to be tested required a positive definite matrix, this matrix was transformed into one with the same

amount of non-zero entries. Then, functions from the different implementations were applied to these matrices and their performance was measured. All elapsed CPU times reported in this manuscript were measured on a single Intel® Xeon® CPU E7-2850 at 2.00 GHz. To quantify the variability of the timing measures, the same function was applied 20 times on the same matrix. The memory usage was measured in terms of peak memory usage, i.e., the maximum amount of MB memory used during the calculations assessed with `gc()` and `gcTorture()`.

Since the measured matrices were relatively small, the `spam64` implementation would never switch to the 64-bit storage format in this setting. Therefore, additional measurements were taken with the `options(spam.force64=TRUE)`, where `spam64` uses the 64-bit storage format in any case.

In Fig. 2, some results for the matrix functions `t()` (transpose), `%*%` (matrix product), `cov()` (calculating the Wendland covariance matrix from a distance matrix) and `chol()` (Cholesky decomposition) are shown. We see that, first, the `spam64` 32-bit storage format has very similar results compared to the `spam` implementation. Second, the `spam64` 64-bit storage format adds a minor overhead because all pointer elements need to be cast from integers to double and vice versa. However, this casting can be easily distributed to multiple processors (task parallelization).

Classically known within the sparse matrix community yet possibly surprising for others, for many operations a significant amount of sparsity is needed to outperform the base implementation. The reduced amount of operations is offset by the handling of the storage structure. For example, replacing the first zero element by an arbitrary number is $O(z)$ for operation count. There is no overarching degree of sparsity when sparse matrices should be used. In addition to operation type, matrix size plays a role.

5. Non-stationary covariance model for a large NDVI residual field

Classical geostatistical models rely on parametric covariance functions to describe the spatial dependency structure of the date. Over the years, many models for anisotropic spatial processes have been proposed (see, e.g., Wackernagel, 2006). Such processes have a translation invariant covariance structure which can be parameterized with a few, typically with five to six, parameters. However, flexible non-stationary models are a quite recent research topic (Kleiber and Nychka, 2012). Most approaches proposed in the literature are very computing intensive and are not suitable for large spatial datasets.

The model proposed in Section 5.2 relaxes the stationarity assumption by allowing the covariance function to depend upon additional covariate data in a parametric way. The model is fitted to a satellite-based vegetation index using elevation data and the distance to the nearest coast as covariates. During the fitting procedure, the new capabilities of `spam64` are used and we get a grasp of the matrix sizes and their associated computation times when dealing with 64-bit integer vectors.

5.1. Data

The availability of long-term satellite earth observations enables the study of changes in the observations at large spatial extents. One primary variable of interest is the normalized difference vegetation

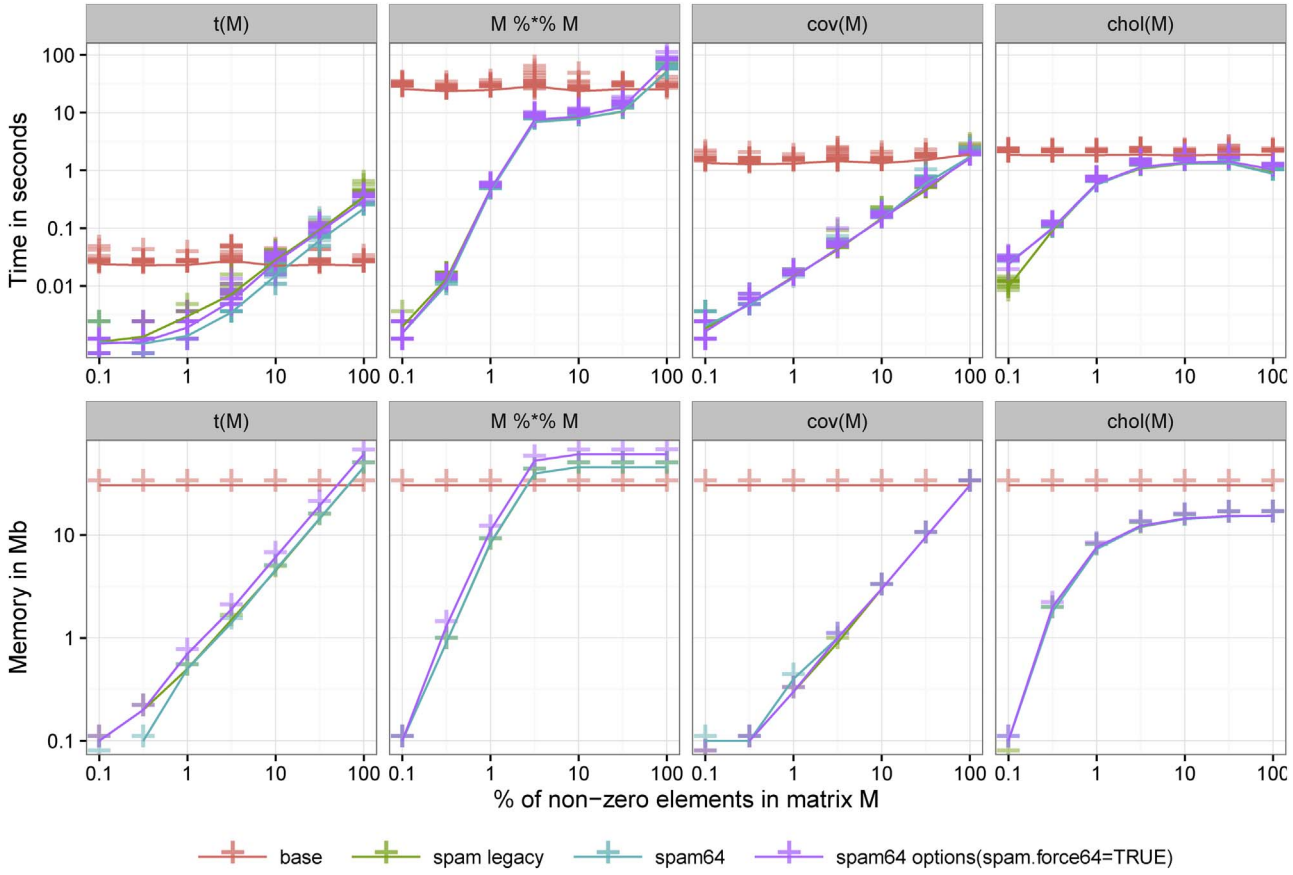

Fig. 2. Elapsed CPU time in seconds and peak memory in Mb for the matrix functions t(M) (transpose), M%*%M (matrix product), cov(M) (Wendland covariance function) and chol(m) (Cholesky decomposition). On the x-axis the % of non-zero elements of the 2000 × 2000 target matrix is indicated.

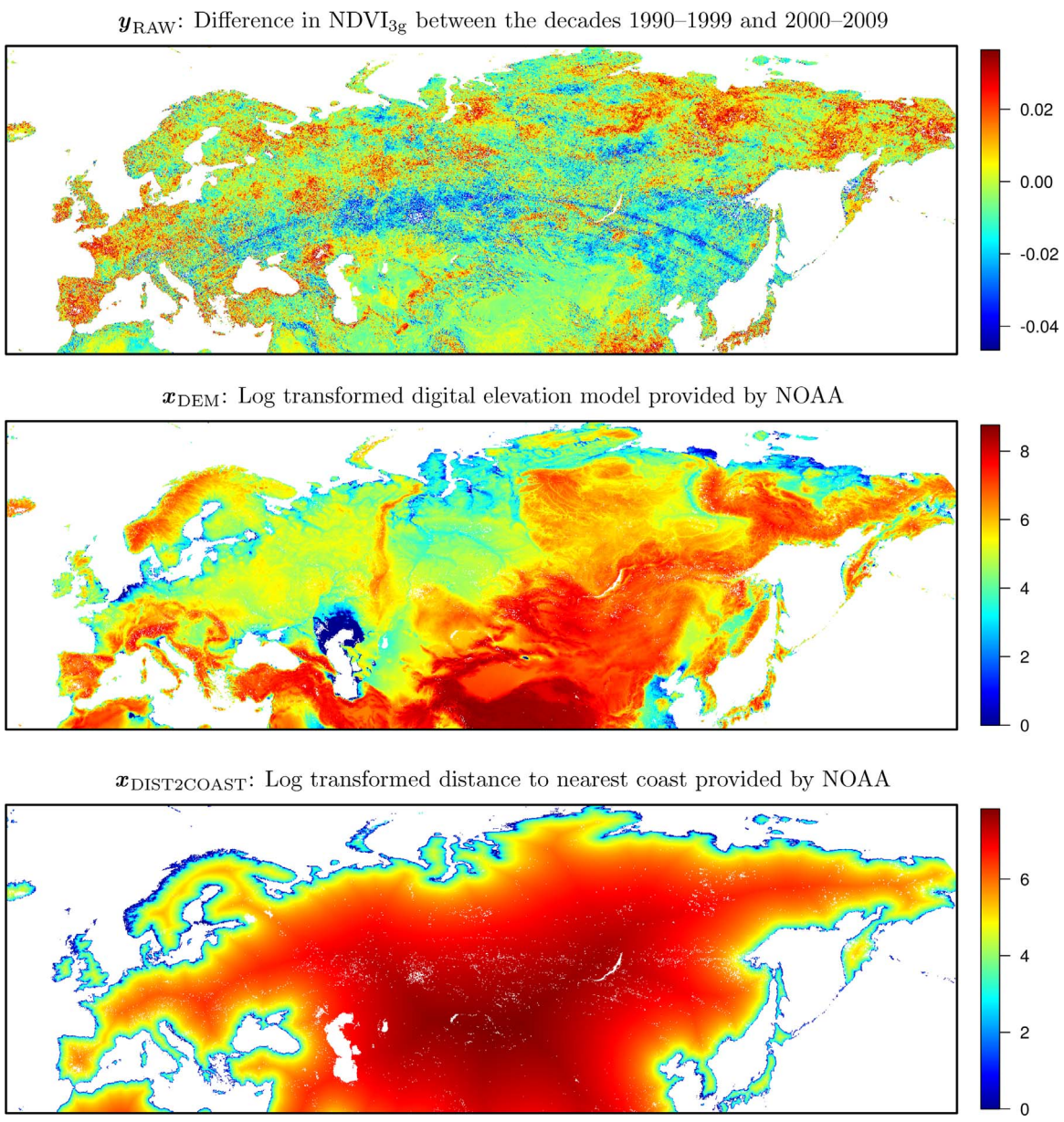

Fig. 3. Data used for the analysis. More precisely, we show y_{RAW} the $NDVI_{3g}$ difference field (top panel), $\mathbf{x}_{DEM}$ the log-transformed digital elevation model data (middle panel) and $\mathbf{x}_{DIST2COAST}$ and the log transformed distance to the nearest coast (bottom panel).

index (NDVI), which serves as a proxy for the intensity of the vegetation through a normalized difference of the near infrared and the red color bands (Myneni and Hall, 1995). We consider the 3rd generation of the Global Inventory Monitoring and Modeling System (GIMMS) NDVI data product ($NDVI_{3g}$), which is a global time series of NDVI data based on information from different Advanced Very High Resolution Radiometer (AVHRR) satellites (Pinzon and Tucker, 2014). The product is available for the years 1981 to 2011 with a temporal resolution of 16 days and is provided on a regular latitude longitude grid with spatial resolution of 1/12° (≈8 km).

To investigate changes in the NDVI in Eurasia between the decades 1990–1999 and 2000–2009 the fields $y_{1990-1999}$ and $y_{2000-2009}$ were created by taking the corresponding 10-year average for each pixel. Then, the difference field was defined as $y_{RAW} = y_{2000-2009} - y_{1990-1999}$. Some pixels of y_{RAW} showed a large difference that was likely to reflect land cover changes from land to water and vice versa. The analysis of these large changes is of interest on its own, but given their large influence on the model fit they should be treated separately. Hence, the lower and the upper 1% quantile of y_{RAW} was removed, leaving n=769,940 observations for the analysis as shown in Fig. 3. To increase numerical stability the mean centered and scaled version of the difference field $y = (y_{RAW} - \bar{y}_{RAW})/\mathrm{sd}(y_{RAW})$ was finally modeled.

We construct the following covariates. First, starting from a 1 km elevation model provided by the National Oceanic and Atmospheric Administration (NOAA) (Hastings et al., 1999) we derived two spatial fields, namely, the logarithm of the elevation denoted as $\mathbf{x}_{DEM}$ and the logarithm of the variability of a 200 km box around the pixel denoted as $\mathbf{x}_{DEM\ VAR}$. Second, the distance to the nearest coast provided by the NOAA's National Ocean Service (oceancolor.gsfc.nasa.gov/DOCS/DistFromCoast/) was log-transformed and is denoted as $\mathbf{x}_{DIST2COAST}$. All three fields were resampled so that we have exactly one observation for each pixel of y. We define $\mathbf{X}$ as the matrix containing the columns $\mathbf{x}_{DEM}$, $\mathbf{x}_{DEM\ VAR}$ and $\mathbf{x}_{DIST2COAST}$. The data preparation and handling was greatly simplified by the R packages `rgdal` (Bivand et al., 2016), `raster` (Hijmans, 2016) and `sp` (Pebesma and Bivand, 2005; Bivand et al., 2013). Figures were made with `ggplot2` (Wickham, 2009) and

`fields` (Nychka et al., 2016).

5.2. Covariance model and implementation

We assume that the NDVI difference field $\boldsymbol{y}$ is a realization of a multivariate normal distribution with mean zero and covariance matrix $\boldsymbol{\Sigma}(\boldsymbol{\theta})$. The covariance depends on the covariates and the parameters $\boldsymbol{\theta} = (\tau, \kappa, \boldsymbol{\beta})^\top$ and has the form

$$\boldsymbol{\Sigma}(\tau, \kappa, \boldsymbol{\beta}) = \kappa \mathbf{D}(\boldsymbol{\beta}) \mathbf{T}(\tau) \mathbf{D}(\boldsymbol{\beta}).$$

where $\kappa > 0$ is a scaling parameter. The diagonal matrix

$$\mathbf{D}(\boldsymbol{\beta}) = \mathrm{diag}(\exp(\mathbf{X}\boldsymbol{\beta})) \tag{1}$$

with $\boldsymbol{\beta} = (\beta_{\mathrm{DEM}}, \beta_{\mathrm{DIST2COAST}}, \beta_{\mathrm{DEM\ VAR}})^\top$ allows $\boldsymbol{\Sigma}$ to spatially vary according to the covariates. The matrix

$$\mathbf{T}(\tau) = (1 - \tau)\mathbf{I} + \tau \mathbf{R}, \quad \tau \in [0, 1]$$

is a weighted average between the identity matrix $\mathbf{I}$ and the correlation matrix $\mathbf{R}$. A similar decomposition was use by Leroux et al. (1999) for the inverse of the covariance matrix, whereas we use it for a substructure of the covariance matrix. In contrast to the classical signal to noise type decomposition, the parameter τ is bounded to the interval $[0, 1]$, which is advantageous for the grid search optimization procedure used later. The matrix $\mathbf{R}$ was calculated via a Wendland (covariance) function (Wendland, 1995; Furrer et al., 2006), with a fixed range of 50 km using the great-circle distance of the spatial locations. Note that because of the use of the great-circle distance and the regular longitude/latitude grid of the data, the number of pixels included in the 50 km range vary with the latitude coordinate as illustrated in the lower panels of Fig. 5. The range parameter of the Wendland function could be estimated from the data, but is fixed here in order to increase the stability of the optimization procedure. The matrix $\mathbf{R}$ (and thus also $\mathbf{T}$) have about $1.28 \cdot 10^8$ non-zero entries (corresponding to about 0.02% of the entire matrix, see also Table 4).

We estimate the parameters $\boldsymbol{\theta}$ with maximum likelihood. Denoting $l(\boldsymbol{\theta}; \boldsymbol{y})$ the log-likelihood of the data we have:

$$-2l(\boldsymbol{\theta}; \boldsymbol{y}) = n \log(2\pi) + \log(\det(\boldsymbol{\Sigma}(\boldsymbol{\theta}))) + \boldsymbol{y}^\top \boldsymbol{\Sigma}(\boldsymbol{\theta})^{-1} \boldsymbol{y}. \tag{2}$$

The computationally expensive log-determinant and quadratic form are expressed as:

$$\log(\det(\boldsymbol{\Sigma})) = n \log(\kappa) + 2 \sum_{i=1}^{n} (\mathbf{X}\boldsymbol{\beta})_i + 2 \sum_{i=1}^{n} \log(\mathrm{chol}(\mathbf{T})_{ii}), \tag{3}$$

$$\boldsymbol{y}\boldsymbol{\Sigma}^{-1}\boldsymbol{y} = \mathbf{v}^\top \mathbf{T}^{-1} \mathbf{v}, \quad \text{where } \mathbf{v} = \mathbf{D}^{-1}\boldsymbol{y}/\sqrt{\kappa}. \tag{4}$$

where $(\mathbf{X}\boldsymbol{\beta})_i$ denoted the ith value of the vector, and $\mathrm{chol}(\mathbf{T})_{ii}$ denotes the ith diagonal entry of the Cholesky decomposition of $\mathbf{T}$. With this decomposition the most time-demanding calculation is the Cholesky decomposition of $\mathbf{T}$, which takes about 30 min on 12 Intel® Xeon® CPUs E7-2850 at 2.00 GHz (see also Table 4). To make the fitting procedure reasonably fast, a grid search was implemented for the parameter τ. More precisely, $\mathrm{chol}(\mathbf{T}(\tau))$ was calculated for $\tau \in S_\tau = \{0, 0.1, 0.2, \ldots, 1\}$ covering the entire range of the parameter space of τ. Then, $-2\,l(\kappa, \boldsymbol{\beta}; \boldsymbol{y}, \tau)$ was minimized for each value of $\tau \in S_\tau$ via the R function `optim()` using the option `method="L-BFGS-B"`, which calls a quasi-Newton optimizer allowing for box constraints (Byrd et al., 1995). Based on these results, a finer grid for τ with a spacing of 0.02 was defined as $S_\tau^{\mathrm{fine}} = \{0.52, 0.54, \ldots, 0.68\}$ and covered the most likely parameter range of τ according to the previous optimization results. Then, $-2\,l(\kappa, \boldsymbol{\beta}; \boldsymbol{y}, \tau)$ was minimized a second time for each value of $\tau \in S_\tau^{\mathrm{fine}}$. The left panel of Fig. 4 shows $\mathrm{argmax}_{\kappa,\boldsymbol{\beta}}\{l(\kappa, \boldsymbol{\beta}; \boldsymbol{y}, \tau)\}$ as a function of the evaluated values of τ. Finally, the value of τ that corresponds to the largest value of $\mathrm{argmax}_{\kappa,\boldsymbol{\beta}}\{l(\kappa, \boldsymbol{\beta}; \boldsymbol{y}, \tau)\}$, together with the configuration of κ and β are reported in Table 3.

For comparison, we also fit a stationary model by setting the diagonal matrix $\mathbf{D}$ to the identity matrix $\mathbf{I}$ in Eq. (1), i.e., $\boldsymbol{\beta} = \mathbf{0}$. Note that $\mathrm{chol}(\mathbf{T}(\tau))$ still depends on τ and therefore the two-step fitting procedure with a gird search for the parameter τ was used again.

5.3. Results and discussion of the model fit

For both covariance function models, the optimizer reported convergence for all values of τ. The estimated parameters and their uncertainty derived from the Hessian matrix are reported in Table 3. Note that due to the grid search for the parameter τ, its value was only estimated up to a resolution of 0.02 and the standard deviation cannot be derived directly. The log-likelihood as a function of the evaluated values of τ is shown for both models in the left panel of Fig. 4.

The estimated values $\boldsymbol{\beta}$ suggest that a smaller elevation and a small distance to the nearest coast result in a larger marginal variances. Furthermore, a small variance in the elevation seems to occur together with small covariance values of $\boldsymbol{y}$. This is in accordance with the observation that the NDVI is sensitive to the occurrence of water (Glenn et al., 2008; Friedl et al., 1995).

To further assess the fit of the models the diagonal elements of $\widehat{\boldsymbol{\Sigma}}$ were considered. For the model with a non-stationary covariance function, their distribution is shown as a histogram in the right panel of Fig. 4. The diagonal value of the stationary model is added as vertical line in the same plot. All values were reasonably close to 1, which is the true value of the variance if the data are modeled as independent observations. The diagonal elements of $\widehat{\boldsymbol{\Sigma}}$ of the non-stationary model are also shown on a map in Fig. 5, giving a visual impression of their spatial distribution. In the same

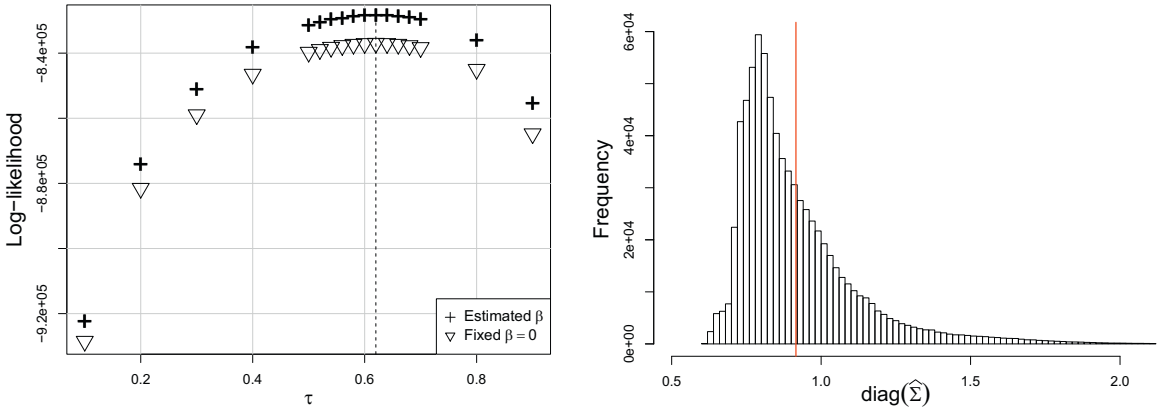

Fig. 4. (left) Log-likelihood for fixed values of τ both for the model with covariate data ($\boldsymbol{\beta}$ is estimated) and without covariate data ($\boldsymbol{\beta} = \mathbf{0}$). (right) Histogram of $\mathrm{diag}(\widehat{\boldsymbol{\Sigma}})$ for the model with covariate data (quartiles: 0.78, 0.86, 0.92 and 0.99). The corresponding value of the model without covariate data is indicated with the red vertical line.

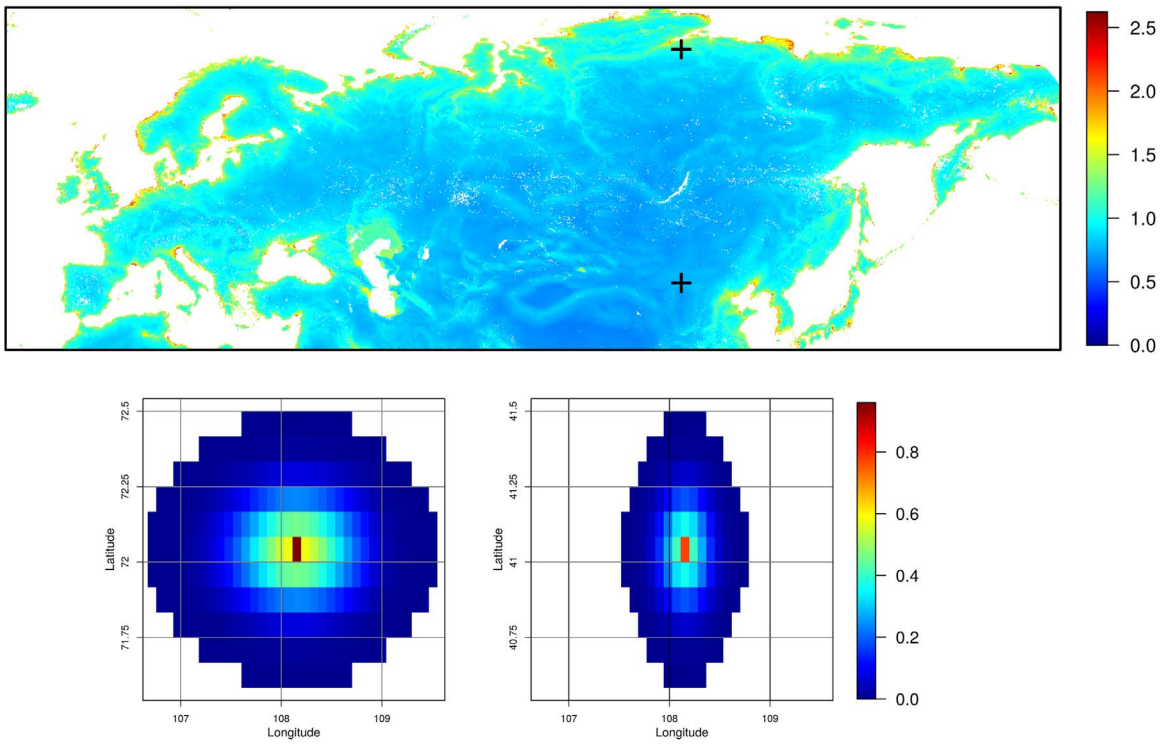

Fig. 5. Diagonal elements of the fitted covariance matrix, i.e., $\mathrm{diag}(\widehat{\boldsymbol{\Sigma}})$ (top). For the two locations indicated with "+" the covariance with the surrounding pixels are shown in the lower panels. Here white areas indicate zero covariance due to its finite support (range) of 50 km. Since the data are stored on a regular latitude longitude grid and the great-circle distance was used, a circle with radius 50 km includes more pixels at the northern location ($N = 292$) compared to the southern location ($N = 121$).

Table 3
Estimates of the parameters τ, κ and $\boldsymbol{\beta}$ for both the non-stationary and the stationary model. For the parameters optimized with the quasi-Newton method the standard errors derived from the Hessian matrix are indicated in parenthesis.

	τ	κ	β_{DEM}	$\beta_{\mathrm{DIST2COAST}}$	$\beta_{\mathrm{DEM\ VAR}}$
Non-stationary	0.62	0.8973	−58.53	−54.38	36.05
		(0.0010)	(0.53)	(0.48)	(0.49)
Stationary	0.62	0.9157	0	0	0
		(0.0010)			

Table 4
Characteristics of the distance matrix **H**, correlation matrix **R** and Cholesky factor $\mathrm{chol}(\mathbf{T}(\tau = 0.62))$ which were used to fit the covariance model in Section 5.2. The `spam` function name used to create the matrix together with the elapsed time, the size, and the density (percentages of non-zero elements) are given. The last column indicates whether 64-bit compiled code was used to generate the matrix.

Matrix	`spam` function	Elapsed time	Size	Density	64-bit
H	nearest.dist()	23.25 Minutes	1.4 Gb	0.02%	–
R	wendland.cov()	1.88 Minutes	1.4 Gb	0.02%	–
chol (**T**)	chol()	29.93 Minutes	8.5 Gb	0.19%	✓

figure the convariance structure for two spatial locations is plotted. It is worth noticing that in terms of the Bayesian information criteria, the more complex model with the non-stationary covariance function provided a better fit (BIC=1,658,282) compared to the model with a stationary covariance function (BIC=1,673,928).

Some characteristics of relevant matrices of the fitting procedure are indicated in Table 4. It is remarkable that only the Cholesky factor actually did use the 64-bit compiled code. However, such matrices can still be handled on a reasonably good desktop computer. From Table 4, we also see that it would take a considerable amount of time to optimize *all* parameters with `optim`, because every evaluation of the likelihood would then require a call to `chol()`. With the grid search approach for the parameter τ we did not only limit the total number of calls to `chol()` but also enabled parallel Cholesky decompositions and optimizations that reduced the fitting time to about 1.5 hours on 12 CPUs as specified above.

6. Discussion

We have illustrated a simple mechanism to extend R packages using the foreign function interface to 64-bit capability. The approach has two fundamental and advantageous benefits: (1) there is no computational overhead in terms of storage and time for small datasets from the end user; (2) the two-package solution is virtually maintenance free; (3) there are only a limited amount of changes in the R code of the original package required. These changes concern the S4 classes, such that these are capable of simultaneously, i.e., appropriately according to the vector length, handling the integers and double.

During this project, the testing phase of the software was disproportionately high. We started to test the functionality of `spam64` in a systematic way using the R package `testthat` (Wickham, 2016, 2011). With the two-package design the amount of tests basically doubles since all functions have to be tested with 32 and 64-bit integers. Actually using 64-bit integers in the testing procedure may increasing testing time quite a bit. However, it may be more safe to do so compared to shortcuts such as using 32-bit integers in conjunction with the `option(spam.force64=TRUE)`. Besides testing functions with respect to the correctness of their returned value, the performance in terms of memory usage and computation time is of great importance when dealing with large objects. Ideally, systematic performance tests in a framework similar to the unit testing are created. This allows the developer to monitor performance impacts of changes in the software

and supports the development of efficient software.

While this article focuses on a practical solution to increase the vector sizes that R can use in combination with compiled code, another aspect of manipulating large vectors is increasing the computation speed at which they are manipulated. The latter can be done by distributing the computational workload through, e.g., MPI (mpi-forum.org/) or OpenMP (www.openmp.org/) and is largely independent of the storage type of the vectors. We experimented with OpenMP to speedup the double to 64-integer castings done by `dotCall64` as described in Section 2. When using OpenMP in conjunction with R, the R package `OpenMPController` (Guest, 2013) was useful to control the number of threads from R. Besides the casting of vectors, there are some Fortran functions in `spam`/`spam64` that should be further optimized with a parallel implementation. Our current focus is the adoption of an efficient parallel Cholesky decomposition, which would enable us to fit the proposed non-stationary covariance model from Section 5 without a grid search for the parameter τ.

It is important to realize that working with huge matrices invokes a tremendous amount of computing time and we reckon that some users might be scared away. Therefore, it is worthwhile spending time install ing an optimized version of R, illustrated by the documentation of `help("long vectors")`: "For example on one particular platform chol on a 47,000 square matrix took about 5 h with the internal BLAS, 21 minutes using an optimized BLAS on one core, and 2 minutes using an optimized BLAS on 16 cores." More specifically, when installing R from its source code, options like `-disable-BLAS-shlib`, `-enable-R-profil-ing`, possibly `-O3` or similar for `CFLAGS` and `FFLAGS`, should be considered. The choice of the linear algebra package (e.g., BLAS (www.netlib.org/blas/), ATLAS (math-atlas.sourceforge.net/), ScaLAPACK (www.netlib.org/scalapack), MKL (software.intel.com/en-us/intel-mkl), openBLAS (www.openblas.net), SuperLU (http://crd-legacy.lbl.gov/~xiaoye/SuperLU/) is important as well. A related discussion can be found in Core Team (2016b).

In Section 2.3 we show an example where the foreign function interface `.Fortran()` is replaced by a call to `.C64()` from package `dotCall64`. The later imitates the `.Fortran()` style but is more efficient (Gerber et al., 2016). It is the authors' believe that the further development and support of a `.Fortran()` style interface is important since it simplifies the integration of compiled code that is not specifically tailored to R. Besides that it is quite popular: Among the 9,079 packages on CRAN (as of 2016-09-02), 12.9% (1,170 packages) make use of the foreign function interface. A comparable proportion use the modern interface to C/C++ (14.6%) and less than 2.1% use both approaches.

The storyline of the article was a spatial data analysis, which was chosen due to current research areas of the authors. We presented a non-stationary covariance model, which is an improvement over using isotropic covariance functions. Fitting the model to the chosen data required storing a Cholesky matrix with more than $2^{31} - 1$ non-zero elements. This was not possible with earlier versions of the R package `spam`, as the used foreign function interface of R does not support long vectors. We showed a way to extend `spam` to work with matrices having more than $2^{31} - 1$ non-zero elements. With that, handling the large Cholesky matrix used for the data analysis became feasible. The data example served as a solid proof of concept for the 64-bit extension strategy. In the area of "big data," there are seemingly countless occasions where manipulating huge vectors with compiled code is required and we are convinced that the move to 64-bit capability is a must that the R community has to address.

Acknowledgments

We thank two anonymous reviewers for their helpful comments. We acknowledge support of the University of Zurich Research Priority Program (URPP) on "Global Change and Biodiversity." We thank the NASA GIMMS team for providing the $NDVI_{3g}$ data for this analysis and Raphael Ostertag for providing the Python code to get the summary statistcs of the foreign function interfaces used by the R packages on CRAN. Furthermore, we thank Rogier de Jong for helpful discussions about the $NDVI_{3g}$ data.

Appendix A. Supplementary data

Supplementary data associated with this article can be found in the online version at http://dx.doi.org/10.1016/j.cageo.2016.11.015.

References

Banerjee, S., Gelfand, A.E., Finley, A.O., Sang, H., 2008. Gaussian predictive process models for large spatial data sets. J. R. Statist. Soc. B 70, 825–848, http://doi.org/10.1111/j.1467-9868.2008.00663.x.

Bevilacqua, M., Gaetan, C., Mateu, J., Porcu, E., 2012. Estimating space and space-time covariance functions for large data sets: a weighted composite likelihood approach. J. Am. Statist. Assoc. 107, 268–280. http://dx.doi.org/10.1080/01621459.2011.646928.

Bivand, R.S., Pebesma, E., Gomez-Rubio, V., 2013. Applied Spatial Data Analysis with R, Second edition. Springer, NY. URL ⟨http://www.asdar-book.org/⟩.

Bivand, R., Keitt, T., Rowlingson, B., 2016. rgdal: Bindings for the Geospatial Data Abstraction Library. URL ⟨http://CRAN.R-project.org/package=rgdal⟩. R package version 1.1-10.

Bivand, R., 2016. CRAN task view: Analysis of spatial data. URL ⟨http://CRAN.R-project.org/view=Spatial⟩. version~2016-09-07.

Byrd, R.H., Lu, P., Nocedal, J., Zhu, C., 1995. A limited memory algorithm for bound constrained optimization. SIAM J. Sci. Comput. 16, 1190–1208. http://dx.doi.org/10.1137/0916069.

R Core Team, 2016a. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing. Vienna, Austria. URL ⟨http://www.R-project.org/⟩.

R Core Team, 2016b. R Installation and Administration. organization R Foundation for Statistical Computing. Vienna, Austria. URL ⟨https://cran.r-project.org/doc/manuals/R-admin.html⟩.

Cressie, N., Johannesson, G., 2008. Fixed rank kriging for very large spatial data sets. J. R. Statist. Soc. B 70, 209–226. http://dx.doi.org/10.1111/j.1467-9868.2007.00633.x.

Eddelbuettel, D., François, R., 2011. Rcpp: seamless R and C++ integration. J. Statist. Softw. 40, 1–18, URL ⟨http://www.jstatsoft.org/v40/i08/⟩.

Eddelbuettel, D., Francois, R., Allaire, J., Ushey, K., Kou, Q., Bates, D., Chambers, J., 2016. Rcpp: Seamless R and C++ Integration 2016.R package version 0.12.7. URL ⟨http://CRAN.R-project.org/package=Rcpp⟩. R package version 0.12.7.

Eddelbuettel, D., 2013. Seamless R and C++ Integration with Rcpp. Springer, New York, URL ⟨http://www.rcpp.org/book/⟩.

Eidsvik, J., Shaby, B.A., Reich, B.J., Wheeler, M., Niemi, J., 2014. Estimation and prediction in spatial models with block composite likelihoods. J. Comput. Graph. Stat. 23, 295–315. http://dx.doi.org/10.1080/10618600.2012.760460.

Eisenstat, S.C., Gursky, M.C., Schultz, M.H., Sherman, A.H., 1977. Yale sparse matrix package I: The symmetric codes. Research Report #112. Yale University, Department of Computer Science.

Friedl, M., Davis, F., Michaelsen, J., Moritz, M., 1995. Scaling and uncertainty in the relationship between the NDVI and land surface biophysical variables: an analysis using a scene simulation model and data from FIFE. Rem. Sens. Environ. 54, 233–246. http://dx.doi.org/10.1016/0034-4257(95)00156-5.

Furrer, R., Genton, M.G., 2011. Aggregation-cokriging for highly-multivariate spatial data. Biometrika 98, 615–631. http://dx.doi.org/10.1093/biomet/asr029.

Furrer, R., Sain, S.R., 2009. Spatial model fitting for large datasets with applications to climate and microarray problems. Stat. Comput. 19, 113–128. http://dx.doi.org/10.1007/s11222-008-9075-x.

Furrer, R., Sain, S.R., 2010. Spam: a sparse matrix R package with emphasis on MCMC methods for Gaussian Markov random fields. J. Stat. Softw. 36, 1–25. http://dx.doi.org/10.18637/jss.v036.i10.

Furrer, R., Genton, M.G., Nychka, D., 2006. Covariance tapering for interpolation of large spatial datasets. J. Comput. Graph. Stat. 15, 502–523, http://dx.doi.org/10.1198/106186006X132178.

Furrer, R., Bachoc, F., Du, J., 2016. Asymptotic properties of multivariate tapering for estimation and prediction. J. Multivariate Anal. 149, 177–191. http://dx.doi.org/10.1016/j.jmva.2016.04.006.

Genton, M.G., 2007. Separable approximations of space-time covariance matrices. Environmetrics 18, 681–695. http://dx.doi.org/10.1002/env.854.

Gerber, F., Furrer, R., 2015. Pitfalls in the implementation of Bayesian hierarchical modeling of areal count data: an illustration using BYM and Leroux models. J. Stat. Softw. 63, 1–32. http://dx.doi.org/10.18637/jss.v063.c01.

Gerber, F., Mösinger, K., Furrer, R., 2016. dotCall64: An efficient interface to compiled C/C++ and Fortran code supporting long vectors. submitted to the R journal.

Glenn, E.P., Huete, A.R., Nagler, P.L., Nelson, S.G., 2008. Relationship between remotely-sensed vegetation indices, canopy attributes and plant physiological processes: what vegetation indices can and cannot tell us about the landscape. Sensors 8, 2136. http://dx.doi.org/10.3390/s8042136.

GNU Fortran compiler, 2014. Reference manual for GCC version 4.9.2. URL ⟨http://gcc.

gnu.org/onlinedocs/gcc-4.9.2/gfortran/⟩.
GNU sed, 2010. Reference manual. URL ⟨http://www.gnu.org/software/sed/manual/⟩.
The GNU C Library, 2014. The GNU C library. URL ⟨http://www.gnu.org/software/libc/manual/html_node/index.html⟩.
Guest, S., 2013. OpenMPController: Control number of OpenMP threads dynamically. URL ⟨http://CRAN.R-project.org/package=OpenMPController⟩. R package version 0.1-2.
Hartman, L., Hössjer, O., 2008. Fast kriging of large data sets with Gaussian Markov random fields. Comput. Stat. Data Anal. 52, 2331–2349. http://dx.doi.org/10.1016/j.csda.2007.09.018.
Hastings, A., D., Dunbar, P.K., Elphingstone, G.M., Bootz, M., Murakami, H., Maruyama, H., Masaharu, H., Holland, P., Payne, J., Bryant, N.A., Logan, T.L., Muller, J.P., Schreierand, G., MacDonald, J.S., 1999. The Global Land One-kilometer Base Elevation (GLOBE) Digital Elevation Model, Version 1.0. National Oceanic and Atmospheric Administration, National Geophysical Data Center, 325 Broadway, Boulder, Colorado 80305-3328, U.S.A. Digital data base on the World Wide Web. URL ⟨http://www.ngdc.noaa.gov/mgg/topo/globe.html⟩.
Hijmans, R.J., 2016. raster: Geographic data analysis and modeling. URL ⟨http://CRAN.R-project.org/package=raster⟩. R package version 2.5-8.
Kaufman, C.G., Schervish, M.J., Nychka, D.W., 2008. Covariance tapering for likelihood-based estimation in large spatial data sets. J. Am. Stat. Assoc. 103, 1545–1555, http://dx.doi.org/10.1198/016214508000000959.
Kleiber, W., Nychka, D., 2012. Nonstationary modeling for multivariate spatial processes. J. Multivariate Anal. 112, 76–91. http://dx.doi.org/10.1016/j.jmva.2012.05.011.
Leroux, B.G., Lei, X., Breslow, N., 1999. Estimation of Disease Rates in Small Areas: A New Mixed Model for Spatial Dependence. IMA Volumes in Mathematics and its Applications, US Government Printing Office. http://dx.doi.org/10.1007/978-1-4612-1284-3_4.
Lindgren, F., Rue, H., Lindström, J., 2011. An explicit link between Gaussian fields and Gaussian Markov random fields: the stochastic partial differential equation approach. J. R. Stat. Soc. B 73, 423–498. http://dx.doi.org/10.1111/j.1467-9868.2011.00777.x.
Mösinger, K., Gerber, F., Furrer, R., 2016. dotCall64: Enhanced Foreign Function Interface Supporting Long Vectors. R package version 0.9-04. URL ⟨http://CRAN.R-project.org/package=dotCall64⟩.
Mösinger, K., 2015. An R implementation for huge spatiotemporal covariance matrices. Master's thesis. University of Zurich.
Myneni, R., Hall, F., 1995. The interpretation of spectral vegetation indexes. IEEE Trans. Geosci. Rem. Sens. 33, 481–486. http://dx.doi.org/10.1109/36.377948.
Nychka, D., Furrer, R., Paige, J., Sain, S., 2016. fields: Tools for Spatial Data. ⟨http://CRAN.R-project.org/package=fields⟩. R package version 8.4-1.
Oehlschlägel, J., 2015. bit64: A S3 class for vectors of 64bit integers. URL ⟨http://CRAN.R-project.org/package=bit64⟩. R package version 0.9-5.
Pebesma, E.J., Bivand, R.S., 2005. Classes and methods for spatial data in R. R News 5, 9–13.
Pebesma, E., 2016. CRAN task view: Handling and analyzing spatio-temporal data. URL ⟨http://CRAN.R-project.org/view=SpatioTemporal⟩. version 2016-08-18.
Pinzon, J.E., Tucker, C.J., 2014. A non-stationary 1981–2012 AVHRR $NDVI_{3g}$ time series. Remote Sensing 6, 6929. URL ⟨http://www.mdpi.com/2072-4292/6/8/6929⟩, http://dx.10.1111/10.3390/rs6086929.
Stein, M.L., Chi, Z., Welty, L.J., 2004. Approximating likelihoods for large spatial data sets. J. R. Stat. Soc. B 66, 275–296. http://dx.doi.org/10.1046/j.1369-7412.2003.05512.x.
Stein, M.L., 2008. A modeling approach for large spatial datasets. J. Korean Stat. Soc. 37, 3–10. http://dx.doi.org/10.1016/j.jkss.2007.09.001.
Sun, Y., Li, B., Genton, M., 2012. Geostatistics for large datasets, in: Porcu, E., Montero, J.M., Schlather, M. (Eds.), Advances and Challenges in Space-time Modelling of Natural Events. Springer Berlin Heidelberg. volume 207 of Lecture Notes in Statistics, pp. 55–77 https://dx.doi.org/10.1007/978-3-642-17086-7_3.
Wackernagel, H., 2006. Multivariate Geostatistics. third ed., Springer-Verlag, New York.
Wendland, H., 1995. Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree. Adv. Comput. Math. 4, 389–396. http://dx.doi.org/10.1007/BF02123482.
Wickham, H., 2009. ggplot2: Elegant Graphics For Data Analysis. Springer, New York, URL ⟨http://www.ggplot2.org/book⟩.
Wickham, H., 2011. testthat: get started with testing. R J. 3, 5–10, URL ⟨https://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf⟩.
Wickham, H., 2016. testthat: Unit Testing for R. URL ⟨https://CRAN.R-project.org/package=testthat⟩. R package version 1.0.2.